SILOON Users Guide

Introduction

SILOON parses your C++ source code and links with your code libraries. As such, you must, of course, provide source code and code libraries! In addition, you must decide the scripting languages for which you'd like interfaces. Currently, Perl and Python are the available choices.

The SILOON system is designed to allow you to have multiple SILOON interfaces. You can create interfaces for multiple unrelated projects, or you may want to partition a single project into several separate and distinct interfaces that can be used independently. We use the term *module* to refer to a single interface in either case. *Note that in the current version of SILOON, you cannot use more than one module in a script. We hope to remove this restriction soon.*

This guide shows you how to initially set up modules and then use modules inside scripts. A running example, save_the_world.cpp, is used to demonstrate how both are done.

Note: These instructions assume that the SILOON package has already been installed in a system location on your machine, along with required software such as the Program Database Toolkit (PDT). See the "INSTALL" file and/or your system administrator for help with these issues.

Sections

- Setting up modules
- Example of setting up a module
- Using modules in scripts
- Example of using a module in a script

---

siloon-team@acl.lanl.gov

Setting up modules

In order to make your C++ libraries available to scripting languages, you must perform the following steps to generate a SILOON interface for your code. These steps must be performed for each module. It doesn't matter what directory you're in when starting (up until step two), but it may be helpful to be in your project's directory.

1. **siloon-init.** Run the `siloon-init` script (located in SILOON's installed bin directory), which takes you through a series of questions asking the module name, location, and scripting languages. It copies a set of standard SILOON files and directories to a directory of your choosing; this directory is called the "module directory" in this document. The information `siloon-init` provides should be self-explanatory.

   `siloon-init` can also be run non-interactively, if you prefer, by specifying all the information on the command line. Run "`siloon-init --help`" to see the command line options.

2. **Change directory.** Change directory to the module directory. You need to be in that directory when performing the following steps.

3. **Compiler flags.** The file `user.defs` in the module directory contains definitions of paths to include files (using the standard "-I" syntax), libraries (using the standard "-L" and "-l" syntax), and object files needed to compile and link with your library. Initially, the file contains only empty definitions. Edit the file and add any required definitions. Paths in the `user.defs` file must be absolute.

4. **siloon-parse.** Run `siloon-parse` (located in SILOON's installed bin directory) on each of the compilation units (e.g. .cpp files) that provide the interface to your code. However, often only one compilation unit is required as interfaces are generated for each included header file. `siloon-parse` will generate a corresponding PDB file with a `.pdb` suffix for each input file. A message will be printed at the end that indicates whether the program was successful or not. If it failed, you will have to stop and try to resolve the problem before going further. This step is very much like a regular compilation, so if your code won't compile, it won't work here, either.

5. **pdbmerge.** If there is only one `.pdb` file from the previous step, you can skip this step. If there is more than one `.pdb` file, run `pdbmerge` (part of PDT), supplying all of the `.pdb` files from the previous step as the input. Use the `-o` flag to specify an output file, which is also a PDB file and therefore should have a `.pdb` suffix.

6. **siloon-gen.** Run `siloon-gen` (located in SILOON's install directory), specifying the single `.pdb` file created in either step 2 or 3. This will generate the following files:
   - `siloon_register.cc`
   - `siloon_execute.cc`
   - `siloon_execute.h`
   - `siloon_includes.h`

of quotes contains a default name generated by SILOON. (This is explained below.) The part between the second set of quotes is the full function prototype.

- `prototypes.excluded` - Functions that have been excluded from the scripting interface, but can be cut and pasted to prototypes.doinclude to be included the next time siloon-gen is run. This file can be quite large, since it will likely include many system functions. The part between the first set of quotes contains a default name generated by SILOON. The part between the second set of quotes is the full function prototype.
- `prototypes.unsupported` - Functions that SILOON is unable to support in this version. They are written purely for your information and cannot be moved to prototypes.doinclude. Only the function prototype is written.
- `prototypes.doinclude` - A user editable file for specifying functions that will be included in the scripting interface the next time siloon-gen is run. Files and directories can also be included, as described above. Please note that the strings in these files must match exactly with SILOON's representation of the prototype, so you should cut and paste from the prototypes.excluded file.

The "module name" files are located in perl or python subdirectories and are generated for each scripting language that you have selected. They contain a set of convenient scripting language "wrapper" functions that call the low level functions that do the real work. The names of these functions are based on the string between the first set of quotes on each line in the `prototypes.doinclude file`. SILOON will generate a default name, which may be mangled if necessary (because of function overloading, template syntax, and other things that make a C++ name unrepresentable in scripting languages). Mangling can generate long, ugly names, so if you don't like the SILOON default, you can put anything you like between the quotes. (Don't edit the prototype inside the second set of quotes.) In the `prototypes.doinclude` example above, because reset is a command in Perl and might cause confusion with a function also named reset, the `reset` member function has had an alias "resetValues" defined for it.

The only file that you should edit is `prototypes.doinclude`. Every other one is generated automatically, so any changes you make will be lost if you re-run siloon-gen.

7. **Build libraries.** Now you need to build the libraries that will be linked into your scripting language(s). How you use your scripting language determines which kind of library to build. If you are using the "standard" approach of creating a script or running the language interactively, then you must build shared libraries; to do that, type `make`. If, however, you are embedding the scripting language into a C or C++ application, then you must build static libraries. To do that, type `make static`.

You're now done. If you make any changes to the interface to your C++ library, you must go through the steps again beginning with `siloon-parse` in order to update the SILOON files. If you need to update information that was provided when you ran `siloon-init`, then you will need to go all the way back to that step.

Example of setting up a module

In this example, a set of simple functions and classes are used to demonstrate some of SILOON's features. These functions and classes are in the files `save_the_world.h` and `save_the_world.cpp`, which are in the directory `/home/user/siloon-example`:

C++

```
// save_the_world.h
//
// Enumeration
//
enum TemperatureType {Cold = -15, LukeWarm = 60, Hot = 110};

//
// Function Definition
//
float temperature(int i, int j);
float temperature(TemperatureType t);

//
// Class Definition
//
class IntegerClass
{
    private: int value;
    public: IntegerClass(int i)                { value = i; }
    public: void setValue(int i)               { value = i; }
    public: int getValue()                     { return value; }
};

//
// Templated Class
//
template <class T> class TClass
{
    private: T value;
    public: TClass(T i)                        { value = i; }
    public: void setValue(T i)                 { value = i; }
    public: T getValue()                       { return value; }
};
```

```
        edgcpfe /home/users/siloon-example/save_the_world.cpp -->
/usr/tmp/303719.il
        taucpdisp -t /usr/tmp/303719.il --> save_the_world.pdb
siloon-parse finished successfully!
```

The next step is to generate the scripting interface files. This is done by running `siloon-gen`, supplying the program database file as input.

```
% siloon-gen save_the_world.pdb
Generating glue code and aliases...
Generating Python wrappers...

Finished.  Inspect the prototypes.included and prototypes.excluded files
to make sure that the required interface has been generated.  If not,
edit prototypes.doinclude as described in the users guide and rerun
siloon-gen.
```

The final step is to build the shared library. This is done by running make.

```
% make
[make output deleted]
```

You are now ready to access the save_the_world code in your Python scripts. The next sections describe how this is done.

<- Setup Intro Usage ->

---

siloon-team@acl.lanl.gov

Using modules in scripts

How the final scripting language library is used varies depending on the language. Much of the functionality is common across every scripting language, so it will be described in general terms first. Later sections describe special instructions or differences for each of the scripting languages.

Perl will be used for examples, but it should be clear what is going on. All code examples are contained in shaded tables, with the particular language used shown at the left of the table.

In the first example, we assume you have a C++ function named "func" defined in your library and that the module is named "example":

C++
```
// A simple function that takes a double and returns an int.
int func(double d) { return(1); }
```

After processing by SILOON, func can be called through the low-level "invoke" function in the scripting language module:

Perl
```
# Call a simple function that takes a double and returns an int.
# $error should still be zero after the call, if all goes well.
# The use of $error is optional.
$aDouble = 4.5;
$error = 0;
$integerReturn = example::invoke("int func(double)"), $aDouble,
                                 $error);
print("The function returned $integerReturn.\n");
print("The error value is $error.\n");
```

If you are using SILOON's wrappers, then you can do the same thing by calling func directly:

Perl
```
# Another way to call a simple function that takes a double and
# returns an int.
$aDouble = 4.5;
$error = 0;
$integerReturn = func($aDouble, $error);
print("The function returned $integerReturn.\n");
print("The error value is $error.\n");
```

Clearly, the wrapper functions are more convenient to use.

C++ objects are created using Perl's standard "new" function. The following example shows how an object of type "Complex" is created and then used.

Python uses different syntax to import modules. Assuming <modulePath> is the full path to your configured module directory and <module> is the name of the module, you must put the following lines at the top of your SILOON Python programs:

Python

```
import sys
# Put the SILOON module location in the include path.
sys.path.insert(0, '<modulePath>/python')
# Import the siloon module.
from <module> import *

# If you don't want to use SILOON's automatically generated
# wrappers, include the following instead of the previous line.
# Even if you do use the wrappers, you can still call invoke directly.
from <module>_siloon import *
```

<- Setup example Intro Usage example ->

---

siloon-team@acl.lanl.gov

Example of using a module in a script

We'll now use the save_the_world example from the setup section in a Perl script. The script will just exercise the functions and methods provided by the save_the_world.o library, which are fairly trivial, but it will show a full, working script.

Perl

```
#!/usr/bin/perl

# save_the_world.pl

# Put the SILOON module location in the include path.
BEGIN {
  push(@INC, "/home/user/siloon-example/siloon-save_the_world
              /perl/blib/arch/auto/save_the_world_siloon");
  push(@INC, "/home/user/siloon-example/siloon-save_the_world/perl");
      };
use save_the_world;

# Call a simple overloaded function.
$temp = temperature_I_I(3, 6);
print("temp = $temp\n");

# Create a new object.  Note that the constructor is mangled because
# it's actually overloaded (the automatically generated copy
# constructor).
$anInt = IntegerClass_I->new(10);

$i = 5;
$anInt->setValue($i);
print("The set value is " . $anInt->getValue() . "\n");
```

The output from running this program follows:

Output

```
temp = 3
The set value is 5
```


siloon-team@acl.lanl.gov